Transparent Rule Based CAS to support Formalization of Knowledge

Reinhard Oldenburg

Abstract. The complexity of computer algebra systems hinders many students to develop an adequate mental model of such a system. As a result, they are often suspicious about the results and the didactical benefit is limited. The paper suggests that it is possible to design a transparent version of a computer algebra system that gives students a transparent access to the inner working of such a system. Didactical uses of such a system are discussed.

Mathematics Subject Classification (2010). Primary 97U70 ; Secondary 68U99.

Keywords. Rule based computation, education.

1. Introduction

Algebra is an important and highly complicated subject of school mathematics that can be structured according to various dimensions of analysis, e.g. it comprises different kinds of activities [3] and structures that obey different linguistic systems [4]. This paper is written in the spirit of that last reference and takes the linguistic view of mathematics to include syntactical, semantical and pragmatic aspects of the algebraic language. Syntactical aspects of algebra can easily be handled by computer algebra systems (CAS). However, there is evidence (e.g. [7] but many others) that students have great difficulty to understand what a CAS does and how its output is to be interpreted.

If a CAS is successfully integrated into students' thinking then two complementary processes should interact: The student learns about the CAS and builds up ideas about its principles. And the students learns about algebra. The next two subsections explore these perspectives.

To use technological artifacts successfully, it is important to develop a mental model [6] of the artifact. Mental models are simplified representations that allow mental operation with the artifact. However, CAS are highly complicated systems and it is hard for students to build up a mental model just from experience with the system. E.g. asking students to interpret and explain CAS results (e.g. why does the system output $1 + x$ when $x + 1$ is entered) usually gives no sensible answers.

On the basis of some knowledge about algebraic manipulation a CAS can also support the learning of algebra, especially it can be used to explore conjectures and to reflect about the power of methods developed so far. On a higher level, students should reflect on what parts of transformational algebra are simply mechanical application of rules and which require more insights.

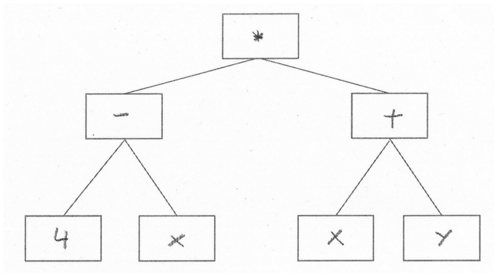
The two learning processes described above benefit from each other and without empirical research it is not yet possible so say how they should be interwoven in an optimal way. The system presented is designed in a way to support both processes.

2. The design of the prototype

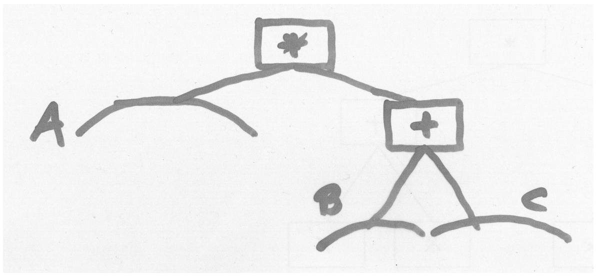
We conducted a small developmental research project that aimed at producing a prototype of a CAS that is as transparent to students as possible. We hope that students can understand completely how this system works.

The system developed is called SCAS (simple syntactical CAS, doubling the S in the acronym was avoided for obvious reasons) and aims to support the learning of syntactical aspects of algebra as well to provide a simple mental model of what a computer algebra system (CAS) is and how it works. The basic use of SCAS is similar to other CAS: You enter an expression and the system answers with an output. E.g. if you enter `2*x+y+x+3` then the system answers `3*x+y+3`.

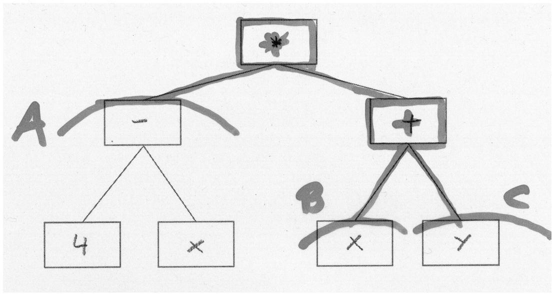
The way how SCAS arrives at this and other answers is based on as little principles as possible. Basically, the students need to understand expressions, patterns and rules. Expressions are binary structures that can easily be represented as trees. The figure shows the expression $(4-x)\cdot(x+y)$.



Patterns are expressions as well but with pattern variables that represent subtrees of expressions. The figure shows the pattern $A\cdot(B+C)$.



Now, if you imagine the expression tree printed on paper and the pattern tree on a transparency (in fact, the image above is from an actual transparency), then you see that the pattern can be put on the expressions and fits its structure allowing to read off values of pattern variables $A=4-x, B=x, C=y$.

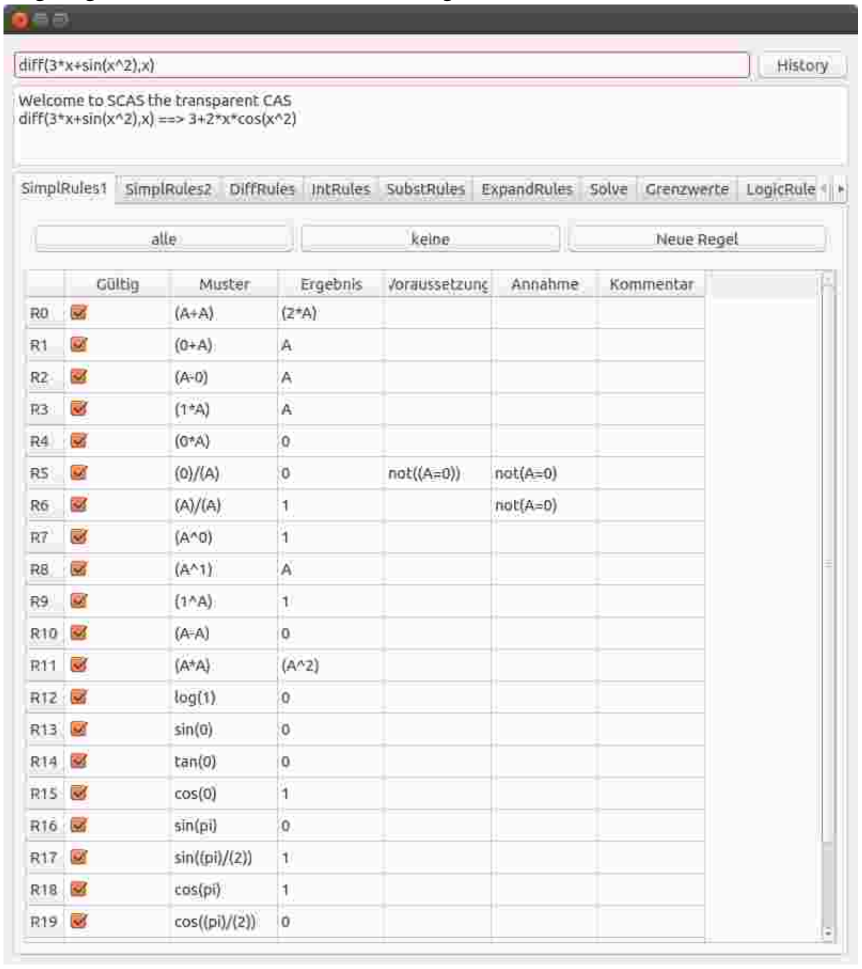


SCAS is a rule based term rewriting system [1]. Of course, modern CAS are not (completely) rule based as many operations have much more efficient algorithmic solutions than those possible by term rewriting. Nevertheless, we believe that this approach gives a good mental model of the syntactic way a CAS treats mathematical expressions.

The following simple rules govern the behavior of the system:

- All expressions are represented by binary trees (e.g. `a+b+c` is interpreted as `a+(b+c)`)
- There is a list of rewrite rules. A rule consists of a pattern, possibly a condition, and a replacement. A handy notation is *pattern→replacement*. The system goes through this list of rules and with each rule it checks if its pattern can be applied to any sub-expression of the current expression. If so it checks if the condition of the rule is satisfied, and if this is the case the expression is replaced.
- This process is repeated until no applicable rule is in the rule list. The final expression is sent to the output.

Students can turn on or off individual rules like `A-A`→ `0` or rule groups (like that for expanding). Moreover, they can enter new rules. The idea is to give them the opportunity to formalize their knowledge e.g. about derivatives. The following screen shot shows the user interface looks like.



```
diff(3*x+sin(x^2),x)
```

```
Welcome to SCAS the transparent CAS
diff(3*x+sin(x^2),x) ==> 3+2*x*cos(x^2)
```

	Gültig	Muster	Ergebnis	Voraussetzung	Annahme	Kommentar
R0	✓	(A+A)	(2*A)			
R1	✓	(0+A)	A			
R2	✓	(A-0)	A			
R3	✓	(1*A)	A			
R4	✓	(0*A)	0			
R5	✓	(0)/(A)	0	not((A=0))	not(A=0)	
R6	✓	(A)/(A)	1		not(A=0)	
R7	✓	(A^0)	1			
R8	✓	(A^1)	A			
R9	✓	(1^A)	1			
R10	✓	(A-A)	0			
R11	✓	(A*A)	(A^2)			
R12	✓	log(1)	0			
R13	✓	sin(0)	0			
R14	✓	tan(0)	0			
R15	✓	cos(0)	1			
R16	✓	sin(pi)	0			
R17	✓	sin((pi)/(2))	1			
R18	✓	cos(pi)	1			
R19	✓	cos((pi)/(2))	0			

2.1. Assumptions made in applying rules

A traditional challenge in designing CAS is the extend to which to restrict to the generic case. E.g. in giving the anti-derivative of x^n as $\frac{x^{n+1}}{n+1}$ one makes the assumption that $n \neq -1$. The situation is tricky if n is on the time of rule application not a number but an expression for which it may be difficult to decide it its value is or can be -1. Many strategies to deal with this problem in special cases or in general have been discussed. Given that SCAS does not try to be the most powerful CAS for a user who needs to arrive quickly at answers, but that is has pedagogical goals, there is a nice way out: When specifying a rule, one has to specify if the rule application is only valid if some condition is met. Such conditions are then collected during rule application and can be displayed in the end. For the anti-derivative example, the rule creator should have written down the condition $n \neq -1$ and in finding the anti-derivative of $x^{sin(k)}$ the system would report the condition $sin(k) \neq -1$, but without solving it.

From a pedagogical point of view, the goal is less to find a way to deal with the problem in the system but to generate in the user consciousness for the problem. Many students lack this consciousness and do not see that $\frac{a^2}{a}$ and a are only equivalent under the assumption $a \neq 0$. Dealing with this problem in SCAS might be a way to solve this problem.

2.2. Limitations

Rule based computation systems have all computational power, they are Turing equivalent. However, a lot of things are cumbersome and inefficient to implement.

In SCAS, is is not so easy to achieve all simplifications one might wish, because many rules are needed to cover all cases. it would be better to reduce the number of operations (e.g. by replacing $A - B$ by $A + (-1) * B$ upon input) and working with n-ary trees to reduce the depth. However, these design decisions would render the rule format more complex. This can be seen in the Mathematica system which has patterns that match a variable number of subexpressions. For SCAS, priority was given to simple rule format at cost of the number of rules that need to be written.

Another difficult point is that infinite loops may easily occur in the rule application sequence. Practically, the system just allows a maximal depth and then goes back, but for the students it is a challenge to understand why and how a certain set of rules combine to produce a cycle.

3. Didactical ideas

A central hope of our approach is that students can use SCAS to formalize their mathematical knowledge by making the system more powerful by giving additional rules. So the idea is to extent Papert's idea of the computer as a trainee to the realm of computer algebra.

In doing this, students experience that the same universally valid equation (e.g. $a \cdot (b + c) = ab + ac$) can be used operationally in two directions to factor or expand expressions. Students will also discover that some transformations require e.g. to divide by an expression, and hence to assume that it is not zero — and they'll find that it is not easy to decide this for symbolic expressions in general.

Formalizing knowledge is a way to make knowledge precise. This may be cumbersome, but we believe that doing so pays off. If this is really the case, should, however, be the topic of further research.

4. Alternatives

Many of the goals presented above can of course be achieved by using some other rule based CAS. Mathematica, e.g. has a very powerful rule based language and one may either define a CAS from scratch or partially use the included CAS algorithms (and this has been used in [5] to explain differentiation). However, this would not make the system as transparent as possible and the complexity of Mathematica's rule language is high. Moreover, it is not as easy as in SCAS to turn off all build-in rules or groups of rules.

Another very easy system is the Pure programming language [2]. This is a rule based language (not a CAS) that works with binary trees as well. The following transcript shows some basic calculations (input after the prompt >, output in the line below), specifying some rules and finally finding derivatives of two functions. Simplification rules are added to teach the system incrementally what is needed to do the task.

```
> 2+3;
5
> k=5;
> k+1;
6
> a;
a
> a+b;
a+b
> a+a;
a+a
> a+a=2*a;
> b+b;
2*b
> x*(y+z)=x*y+x*z;
> 2*(a+4);
2*a+8
> b*(a+4);
b*a+b*4
> b*(a+0);
b*a+b*0
> a*0=0;
> 0*a=0;
> a+0=a;
> b*(a+0);
b*a
> diff(x,x)=1;
> diff(x^n,x)=n*x^(n-1);
> diff(u^n,x)=n*u^(n-1)*diff(u,x);
> diff(u+v,x)=diff(u,x)+diff(v,x);
> diff(4*x^3+2*x^2+x,x);
diff (4*x^3,x)+diff (2*x^2,x)+1
> diff(u*v,x)=v*diff(u,x)+u*diff(v,x);
> diff(4*x^3+2*x^2+x,x);
x^3*diff (4,x)+4*(3*x^2)+(x^2*diff (2,x)+2*(2*x^1))+1
> diff(y::int,x)=0;
> diff(4*x^3+2*x^2+x,x);
0+4*(3*x^2)+(0+2*(2*x^1))+1
> 0+x=x;
> x^1=x;
> diff(4*x^3+2*x^2+x,x);
4*(3*x^2)+2*(2*x)+1
> (x::int)*(y::int*z)=(x*y)*z;
> diff(4*x^3+2*x^2+x,x);
12*x^2+4*x+1
> diff(sin(u),x)=cos(u)*diff(u,x);
> diff(x^2*sin(x^2+1),x)
```

```
2*x^1*sin (x^2+1)+x^2*(cos (x^2+1)*(2*x^1))
> a^1=a;
2*x*sin (x^2+1)+x^2*(cos (x^2+1)*(2*x))
```

We feel, however, that Pure is not so easy to master for students as it is not so easy to turn off or on individual rules and once a bad rule is entered, the system may behave strangely and require starting the interpreter again.

5. Conclusion

This paper leaves open many questions. Up to know, there is no experience working with students. It is not so easy to do this, because German curricula leave little room for experimentation. Nevertheless, we think that the approach is worth being investigated further.

References

[1] F. Baader, T. Nipkow, *Term Rewriting and All That*, Cambridge UP, Cambridge 1999.

[2] A. Graef: Pure. http://purelang.bitbucket.org/

[3] C. Kieran, *The core of algebra: reflections on its main activities.*, in K. Stacey et al. (eds.), The future of the teaching and learning of algebra. Kluwer Academic Publishers, Dordrecht (ISBN 1-4020-8130-8), pp. 21-33 (2004).

[4] J. Hodgen, D. Kuechemann, R. Oldenburg, *Syntactic and Semantic Items in Algebra Tests - A conceptual and empirical view*, CERME, 2013.

[5] W. Köpf: Computeralgebra. Springer, Berlin (2006).

[6] P. Johnson-Laird, *Mental models. Towards a cognitive science of language, inference, and consciousness.*, 6th print, Harvard Univ. Press, Cambridge 1995.

[7] R. Oldenburg, B. Weygandt, *Einsatzmöglichkeiten und Grenzen von Computeralgebrasystemen zur Förderung der Konzeptentwicklung*, in A. Hoppenbrock et al. (eds.) Lehren und Lernen von Mathematik in der Studieneingangsphase. Springer, Berlin (2015).

Reinhard Oldenburg
Universitätsstraße 14
D-86159 Augsburg
Germany
e-mail: `reinhard.oldenburg@math.uni-augsburg.de`